**Exercise-01:**

### 1. Understand the Problem

When developing an inventory management system, it's important to realize the role data structures and algorithms play in managing large amounts of data efficiently. A warehouse typically contains hundreds or thousands of products, each with unique identifiers and associated information like quantity and price. Efficient data storage and retrieval ensure that operations such as adding, updating, and deleting products can be done quickly without degrading system performance. This is where the choice of data structures becomes crucial.

To manage such inventory effectively, suitable data structures must support fast access and updates. ArrayList and HashMap are commonly used in such scenarios. An ArrayList is simple and maintains insertion order, making it good for displaying lists. However, for fast lookup based on unique product identifiers like product IDs, a HashMap is more suitable because it provides constant time complexity for operations like insert, search, and delete on average.

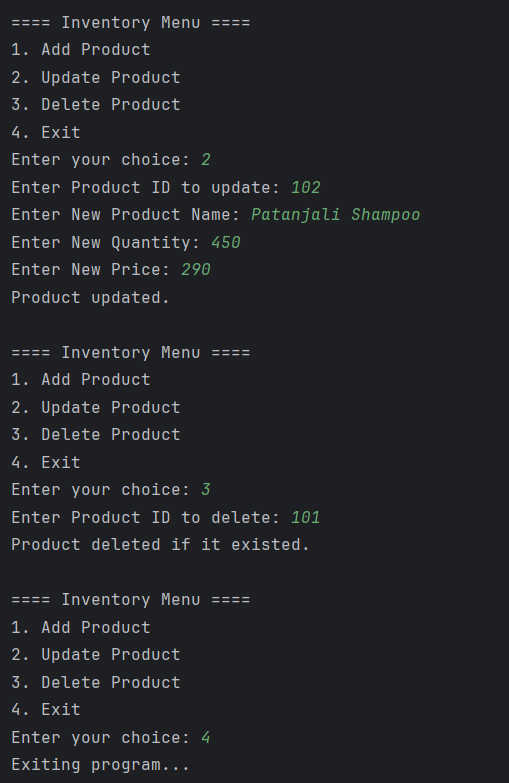
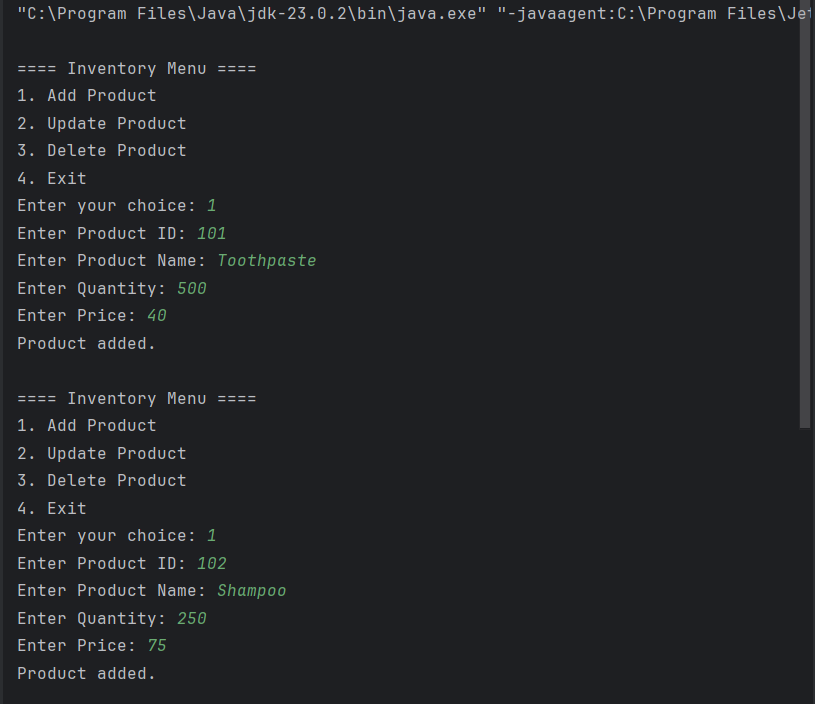
### 2. Analysis

Using a HashMap to store products provides efficient operations. The time complexity for adding a product is O(1) on average, since inserting into a HashMap is constant time. Updating a product also takes O(1) time because the product can be retrieved by its ID instantly and modified directly. Deleting a product is similarly efficient, with an average time complexity of O(1) as the entry is removed using the product ID as the key.

To further optimize the system, an additional HashMap can be introduced for indexing by product name. This allows quick lookup of products not just by ID but also by name, which is useful when multiple users or systems might not know the product ID but remember the product name. For large inventories or scenarios where data needs to persist beyond program execution, integrating a relational database is a practical solution. It enables persistent storage, better data management, and scalability for real-world deployment. Batch operations can also be introduced to handle large inputs, such as uploading data from files, improving performance by reducing repetitive individual insertions.

In conclusion, the implementation makes effective use of HashMap for efficient inventory operations and can be further extended for more complex requirements with minor architectural changes.

**3. Output**

****

**Exercise-02:**

### 1. Understand Asymptotic Notation

When analyzing the performance of algorithms, asymptotic notation such as Big O is used to describe how the runtime of an algorithm grows as the size of the input increases. Big O notation focuses on the upper bound of an algorithm's runtime and gives a worst-case scenario. This helps developers choose algorithms that remain efficient even as data grows larger.

For search operations, the best case is when the item is found at the beginning of the collection, the worst case is when the item is not found or is at the end, and the average case lies somewhere in between. Understanding these cases is essential in designing a high-performing search feature for platforms like e-commerce websites where large product catalogs are involved.

### 2. Analysis

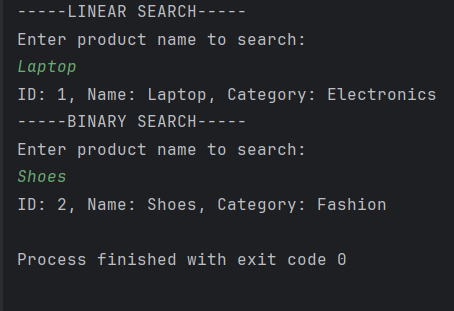
Linear search has a time complexity of O(n), where n is the number of elements in the array. This means that in the worst case, every element must be checked, which is not efficient for large datasets. However, linear search is simple and does not require sorting.

Binary search has a time complexity of O(log n) due to its divide-and-conquer approach, which makes it significantly faster for large datasets. The trade-off is that the data must be sorted before performing the search, which itself takes O(n log n) time using efficient sorting algorithms.

In the context of an e-commerce platform where performance and scalability are important, binary search is more suitable, especially when product data can be pre-sorted or when searches are frequent. Sorting the dataset once and performing multiple binary searches afterward will result in better overall performance.

Therefore, for a small or unsorted product list, linear search is acceptable. For large datasets where quick search results are necessary, binary search combined with efficient sorting is the preferred approach.

**3. Output**



**Exercise-03:**

### 1. Understand Sorting Algorithms

Sorting algorithms are techniques used to reorder data in a particular format, such as ascending or descending. They are crucial in applications like e-commerce, where data like customer orders need to be displayed based on price, date, or priority.

Bubble Sort is a simple comparison-based algorithm where adjacent elements are compared and swapped if they are in the wrong order. It repeats this process until the entire list is sorted. It is easy to understand and implement but inefficient for large datasets due to its high time complexity.

Insertion Sort builds the sorted array one element at a time. It is more efficient than Bubble Sort for small or nearly sorted data but still not suitable for large datasets.

Quick Sort is a more efficient divide-and-conquer algorithm. It selects a pivot element and partitions the array so that all smaller elements go to the left and larger ones to the right. It then recursively applies the same process to each subarray. Quick Sort is known for its speed and efficiency in most practical scenarios.

Merge Sort is another divide-and-conquer algorithm that splits the array into halves, recursively sorts each half, and then merges the sorted halves. It has consistent performance but uses extra memory due to the merging process.

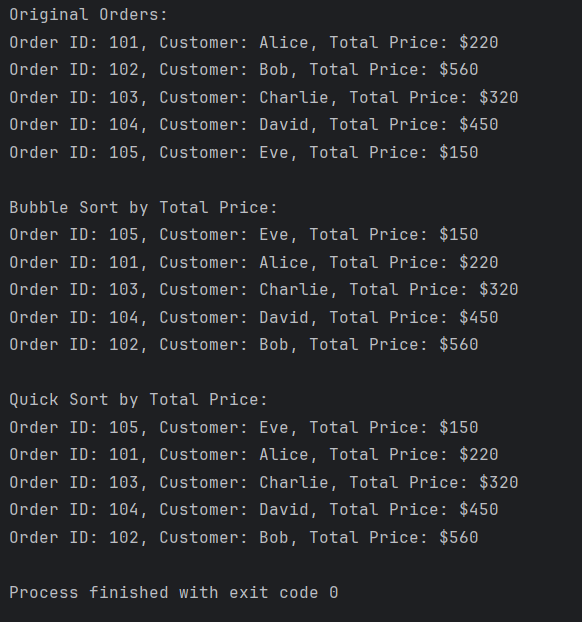
### 2. Analysis

Bubble Sort has a time complexity of O(n²) in the worst and average case. It performs poorly on large datasets because it requires multiple passes through the array and performs many comparisons and swaps.

Quick Sort, on the other hand, has an average-case time complexity of O(n log n), which is much faster for large datasets. Although its worst-case complexity is O(n²), this rarely occurs with good pivot selection or randomization. Quick Sort is typically preferred in production systems because of its efficiency and low memory usage compared to algorithms like Merge Sort.

In the context of an e-commerce platform where hundreds or thousands of orders may need to be sorted based on price, Quick Sort is the better choice. It provides significantly faster performance and scales better as the number of orders increases.

**3. Output**

****

**Exercise-04:**

### 1. Understand Array Representation

Arrays are a fundamental data structure used to store a fixed number of elements of the same type in contiguous memory locations. This means each element can be accessed directly using an index, allowing for fast retrieval in constant time. The memory layout ensures that accessing elements by position is efficient, as the location of each element can be calculated using a formula.

Arrays are advantageous because they offer fast random access and low overhead, making them useful for storing data that doesn't change frequently. However, they have a fixed size, and resizing them requires creating a new array and copying data, which can be inefficient.

### 2. Analysis

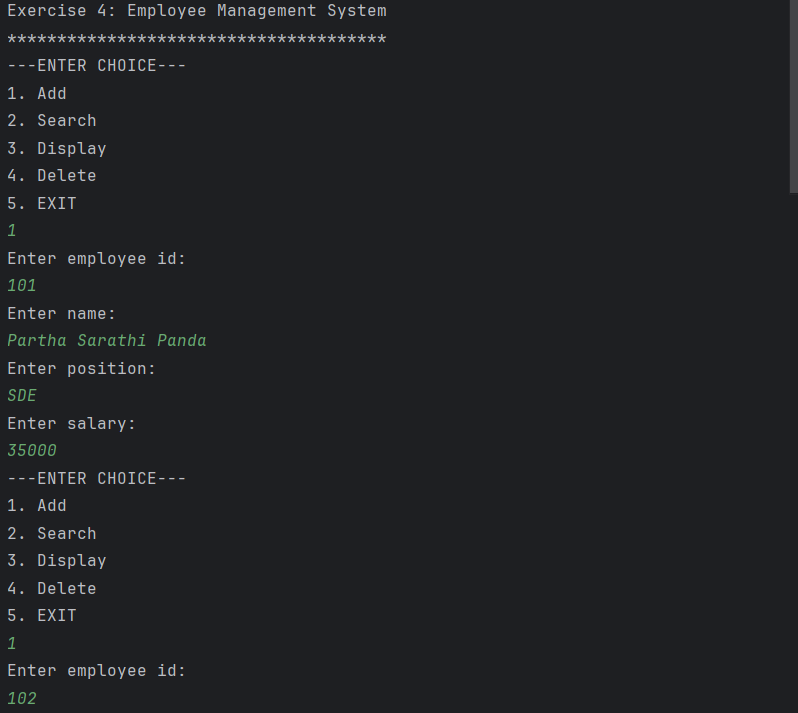
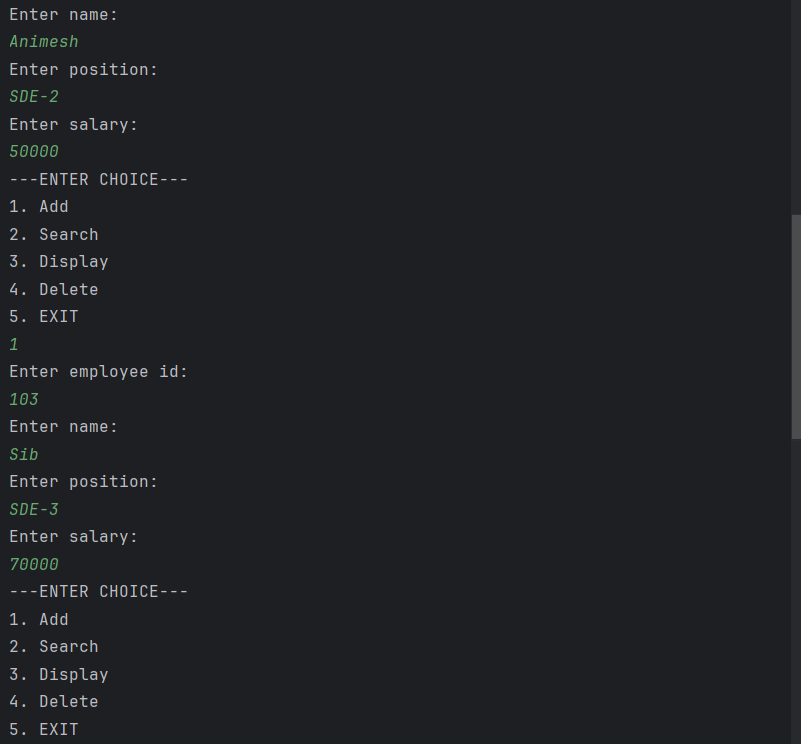
The time complexity of each operation using an ArrayList (or array) is as follows:

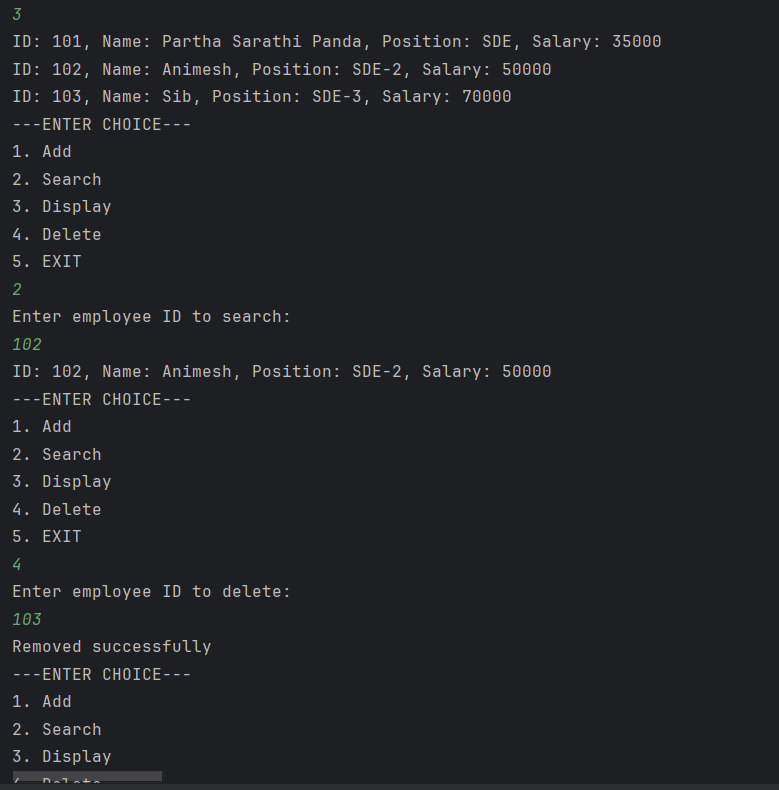
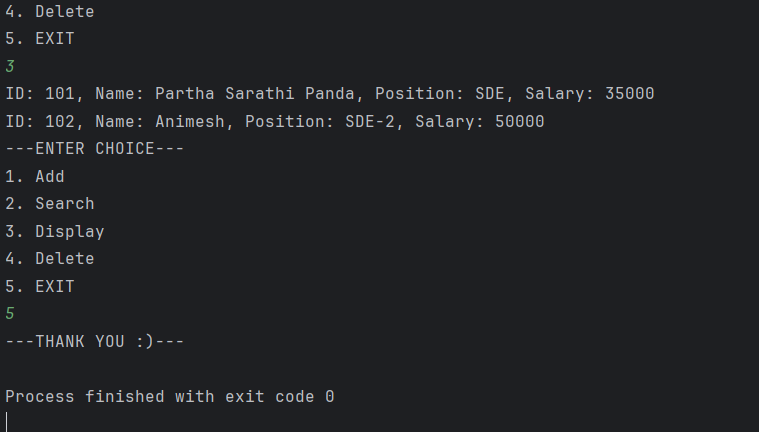
* **Add**: O(1) on average, since adding to the end of an ArrayList is typically constant time.
* **Search**: O(n), as each element must be checked until a match is found.
* **Traverse**: O(n), because all elements are printed one by one.
* **Delete**: O(n), since the element must first be found and then removed, which may involve shifting elements in memory.

While arrays are efficient for accessing elements by index, they are not ideal when frequent insertions and deletions are needed at arbitrary positions. In such cases, using data structures like LinkedList or HashMap may offer better performance depending on the use case.

Arrays or ArrayList should be used when the number of elements is moderate, the data structure is mostly read-heavy, or when index-based access is required. For large or frequently modified datasets, other structures like maps or trees may be more suitable.

**3. Output**

** **

** **

**Exercise-05:**

**1. Understand Linked Lists**

A linked list is a linear data structure where elements, known as nodes, are connected using pointers. Each node contains data and a reference to the next node in the sequence.

There are mainly two types of linked lists:

* A **Singly Linked List** allows traversal in one direction. Each node stores a reference to the next node only.
* A **Doubly Linked List** supports traversal in both directions. Each node maintains two references — one for the next node and another for the previous node.

Linked lists are dynamic, meaning they can grow or shrink in size at runtime without the need for resizing or reallocation, unlike arrays.

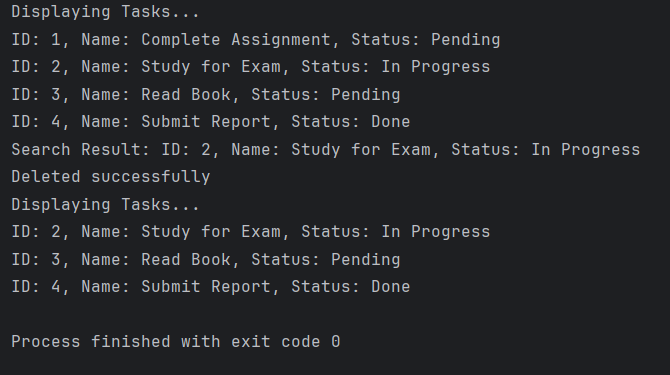
**2. Analysis**

The time complexity of operations on a singly linked list is as follows:

* **Add**: O(n) — since we must traverse to the end to insert the new node.
* **Search**: O(n) — because each node must be checked one by one.
* **Traverse**: O(n) — each node is visited once.
* **Delete**: O(n) — similar to search, the node must first be located, and then pointers are updated.

Although these operations have linear time complexity, the **major advantage of a linked list** is its flexibility in memory usage and dynamic resizing. Unlike arrays, linked lists do not require a predefined size and do not involve shifting elements during insertion or deletion. This makes them ideal for applications like task management systems where the number of tasks can change frequently.

**3. Output**

****

**Exercise-06:**

**1. Understand Search Algorithms**

Search algorithms are fundamental for retrieving information from a dataset. Two common techniques are **linear search** and **binary search**.

In a **linear search**, each element in the list is checked sequentially until the target is found or the list ends. It does not require the data to be sorted and works on any kind of list.

In contrast, **binary search** is a more efficient algorithm that repeatedly divides a sorted list into halves to locate the target. It starts by comparing the middle element with the target. If the target is smaller, the search continues in the left half; otherwise, it proceeds in the right half. Binary search only works on sorted lists and is significantly faster for large datasets.

**2. Analysis**

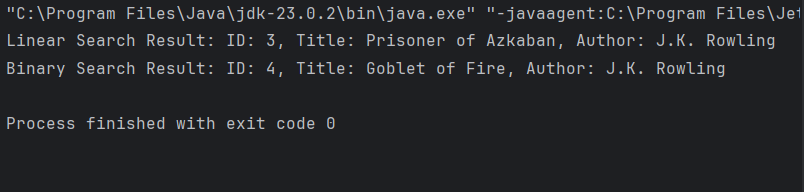
* **Linear Search Time Complexity**: O(n), where *n* is the number of books. It checks each book one by one and is suitable for small datasets or unsorted data.
* **Binary Search Time Complexity**: O(log n), where *n* is the number of books. It is much faster than linear search for large and sorted datasets.

The choice between linear and binary search depends on the dataset:

* Use **linear search** when the data is small or unsorted.
* Use **binary search** when the dataset is large and sorted.

In the context of a library management system, binary search is ideal when books are sorted by title, especially for fast retrieval. However, linear search remains useful when real-time sorting isn’t possible or when searching by other attributes that aren't sorted.

**3. Output**

****

**Exercise-07:**

### 1. Understand Recursive Algorithms

Recursion is a programming technique where a function calls itself to solve smaller subproblems of the original task. It is particularly useful when a problem can be broken down into similar subproblems. Recursive algorithms simplify the code by reducing complex iterative logic into concise, self-referential definitions.

In financial forecasting, recursion can be applied to predict future values based on previous data trends, such as compound growth, interest accumulation, or year-over-year changes. Each forecasted value depends on the outcome of the previous one, making recursion a natural fit.

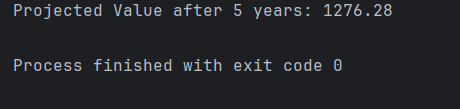
### 2. Analysis

The time complexity of a simple recursive forecasting function is **O(n)**, where n is the number of years to forecast. For each year, one recursive call is made until the base case is reached.

However, recursion may become inefficient if the problem involves overlapping subproblems or a large number of recursive calls. In such cases, **memoization** or **converting the recursive solution to an iterative one** can significantly reduce computation time.

For this problem, since each result is only calculated once and depends directly on the previous one, the recursion remains efficient and linear in complexity. But in more complex scenarios (e.g., if values depended on multiple previous years), dynamic programming techniques would be necessary to optimize performance and avoid redundant calculations.

**3. Output**

****